FindBug安装及使用说明

**目录**

[1. 简介 3](#_Toc338773338)

[1.1 说明 3](#_Toc338773339)

[1.2 环境要求 3](#_Toc338773340)

[1.3 注意事项 3](#_Toc338773341)

[2. FindBugs安装 3](#_Toc338773342)

[2.1 在线安装 3](#_Toc338773343)

[2.2 离线安装 9](#_Toc338773344)

[3. FindBugs使用 10](#_Toc338773345)

[3.1 FindBugs操作 10](#_Toc338773346)

[3.2 FindBugs常见故障模式 11](#_Toc338773347)

[3.2.1 NP\_NULL\_ON\_SOME\_PATH: Possible null pointer dereference 11](#_Toc338773348)

[3.2.2 DLS\_DEAD\_LOCAL\_STORE: Dead store to local variable 11](#_Toc338773349)

[3.2.3 BX\_BOXING\_IMMEDIATELY\_UNBOXED: Primitive value is boxed and then immediately unboxed 12](#_Toc338773350)

[3.2.4 DM\_NUMBER\_CTOR: Method invokes inefficient Number constructor; use static valueOf instead 12](#_Toc338773351)

[3.2.5 Cloneable Not Implemented Correctly (CN) 12](#_Toc338773352)

[3.2.6 OBL\_UNSATISFIED\_OBLIGATION: Method may fail to clean up stream or resource 12](#_Toc338773353)

[3.2.7 Double Checked Locking (DC) 13](#_Toc338773354)

[3.2.8 Dropped Exception (DE) 14](#_Toc338773355)

[3.2.9 Suspicious Equals Comparison (EC) 14](#_Toc338773356)

[3.2.10 Bad Covariant Definition of Equals (Eq) 15](#_Toc338773357)

[3.2.11 Equal Objects Must Have Equal Hashcodes (HE) 15](#_Toc338773358)

[3.2.12 Static Field Modifiable By Untrusted Code (MS) 15](#_Toc338773359)

[3.2.13 Null Pointer Dereference (NP), Redundant Comparison to Null (RCN) 16](#_Toc338773360)

[3.2.14 Non-Short-Circuit Boolean Operator (NS) 17](#_Toc338773361)

[3.2.15 Open Stream 17](#_Toc338773362)

[3.2.16 Read Return Should Be Checked (RR) 17](#_Toc338773363)

[3.2.17 Return Value Should Be Checked (RV) 18](#_Toc338773364)

[3.2.18 Non-serializable Serializable class (SE) 18](#_Toc338773365)

[3.2.19 Uninitialized Read In Constructor (UR) 18](#_Toc338773366)

[3.2.20 Unconditional Wait (UW) 19](#_Toc338773367)

[3.2.21 Wait Not In Loop (Wa) 19](#_Toc338773368)

# 简介

## 说明

FindBugs 是一个java bytecode静态分析工具，它可以帮助java工程师提高代码质量以及排除隐含的缺陷。

FindBugs检查类或者 JAR 文件，将字节码与一组缺陷模式进行对比以发现可能的问题。

有了静态分析工具，就可以在不实际运行程序的情况对软件进行分析。FindBugs不是通过分析类文件的形式或结构来确定程序的意图，而是通常使用 Visitor 模式进行分析。（[Visitor 模式的更多信息](http://en.wikipedia.org/wiki/Visitor_pattern)）

## 环境要求

使用FindBugs至少需要JDK1.4.0以上版本，FindBugs是平台独立的，可以运行于GNU/Linux、Windows、MacOS X　等平台上。  
 运行FindBugs至少需要有256 MB内存，如果你要分析一个很大的项目，那就需要更加多的内存。

## 注意事项

本文部分内容来自网络，如有错误或不当之处请指出，谢谢。

# FindBugs安装

## 在线安装

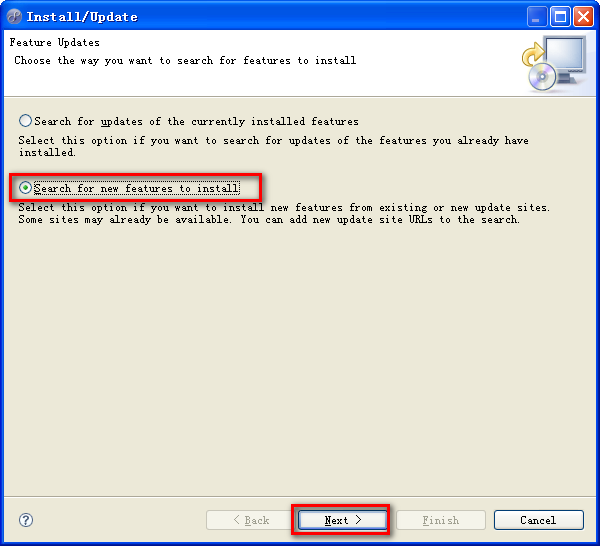
使用Eclipse/MyEclipse的Help🡪Software Updates🡪Find and Install以插件形式在线安装FindBugs。请安装的同学注意网络是否正常，建议17：00-19：00时间段操作。  


图2-1-1 查找新的插件进行安装

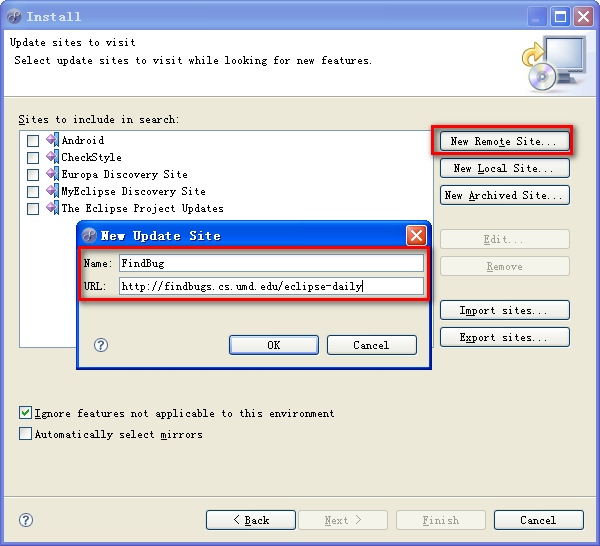


图2-1-2 新建findbugs的远程站点

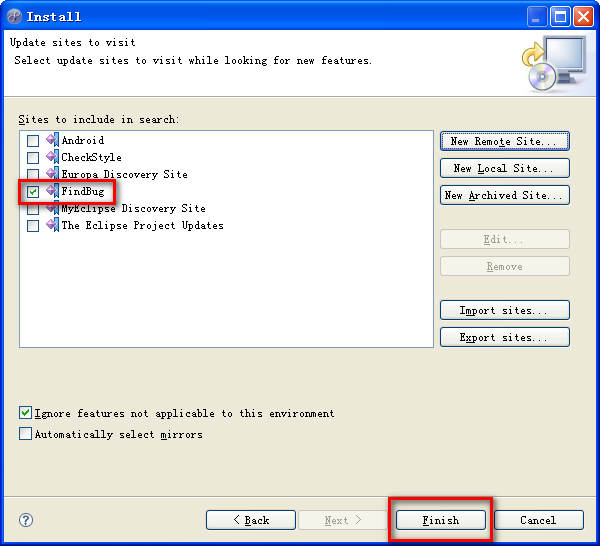


图2-1-3 勾选配置的站点，完成开始远程下载最新的FindBugs插件包

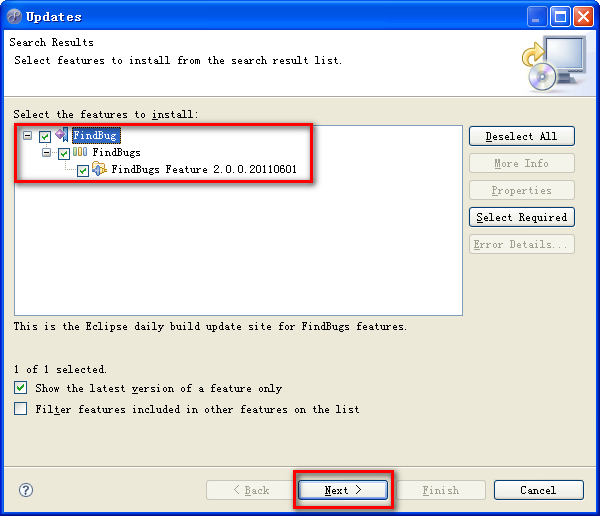


图2-1-4 勾选下载的插件包，进行下一步

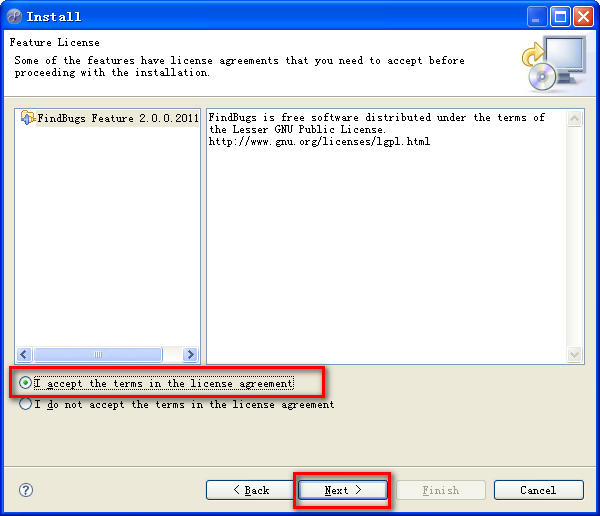


图2-1-5 勾选接受选项，进行下一步

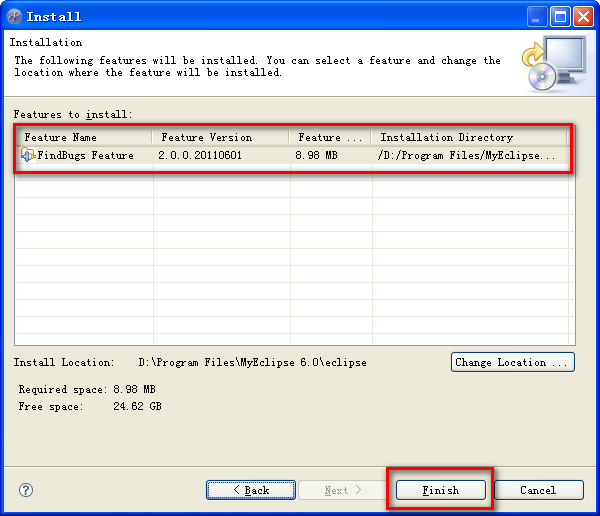


图2-1-6 选择完成，进行FindBugs插件安装

## 离线安装

请到<http://findbugs.sourceforge.net/downloads.html>下载Eclipse plugin for FindBugs，目前版本为1.3.9.20090821 。

将下载的edu.umd.cs.findbugs.plugin.eclipse\_1.3.9.20090821.zip文件解压到Eclipse的plugins子目录中，重新启动Eclipse即可。

<ftp://172.16.6.1:22/>[部门/统一开发环境/IDE/JAVA/eclipse plugin/findbugs-1.3.9.zip](ftp://172.16.6.1:22/部门/统一开发环境/IDE/JAVA/eclipse%20plugin/findbugs-1.3.9.zip)

解压运行findbugs.bat,

Eclipse插件使用

安装插件edu.umd.cs.findbugs.plugin.eclipse\_1.3.9.20090821.zip

[ftp://172.16.6.1:22/部门/统一开发环境/IDE/JAVA/eclipse plugin/edu.umd.cs.findbugs.plugin.eclipse\_1.3.9.20090821.zip](ftp://172.16.6.1:22/部门/统一开发环境/IDE/JAVA/eclipse%20plugin/edu.umd.cs.findbugs.plugin.eclipse_1.3.9.20090821.zip)

# FindBugs使用

## FindBugs操作

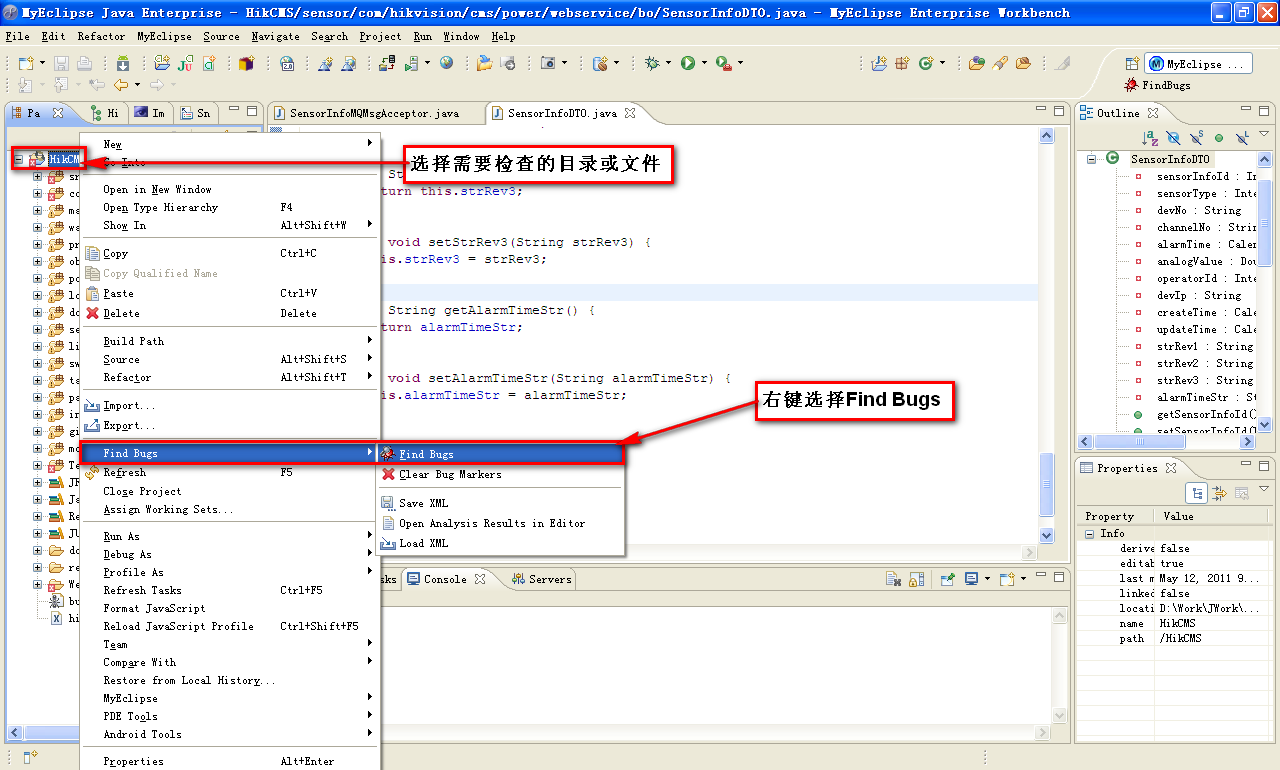


图3-1-1 选择检查的内容，右键Find Bugs开始进行代码检查

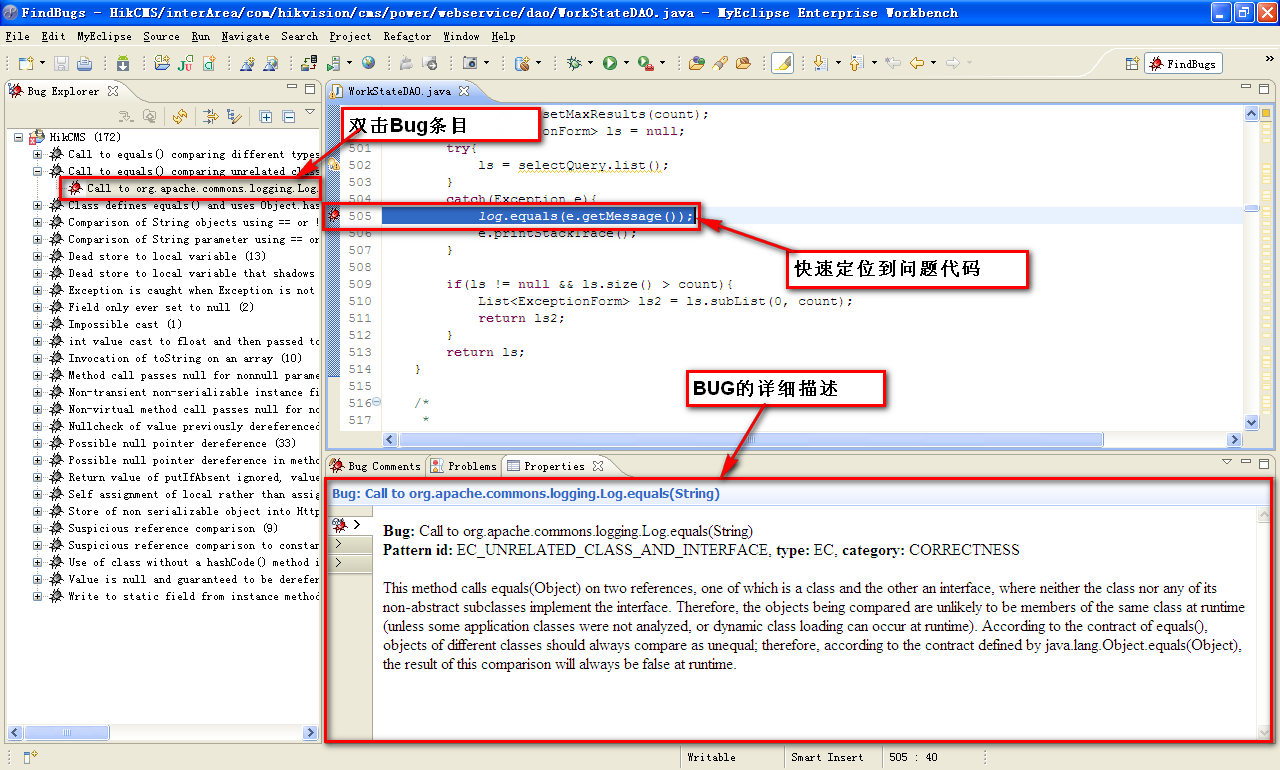


图3-1-2 检查完毕切换到FindBugs透视图查看Bug条目

## FindBugs常见故障模式

### NP\_NULL\_ON\_SOME\_PATH: Possible null pointer dereference

There is a branch of statement that, if executed, guarantees that a null value will be dereferenced, which would generate a NullPointerException when the code is executed. Of course, the problem might be that the branch or statement is infeasible and that the null pointer exception can't ever be executed; deciding that is beyond the ability of FindBugs.

这是可能导致空指针异常的一种潜在问题。

例如：if (type.equals(“gdbb”))应改成if (“gdbb”.equals(type))

### DLS\_DEAD\_LOCAL\_STORE: Dead store to local variable

This instruction assigns a value to a local variable, but the value is not read or used in any subsequent instruction. Often, this indicates an error, because the value computed is never used.

无用变量的问题，许多变量定义了但是没有使用过

### BX\_BOXING\_IMMEDIATELY\_UNBOXED: Primitive value is boxed and then immediately unboxed

A primitive is boxed, and then immediately unboxed. This probably is due to a manual boxing in a place where an unboxed value is required, thus forcing the compiler to immediately undo the work of the boxing.

封装之后立马拆箱

例如：int a = 3;

double d = Double.valueOf(a).doubleValue/4;

要换成double d = (double)a/4;或者double c = a;double d = c/4;

### DM\_NUMBER\_CTOR: Method invokes inefficient Number constructor; use static valueOf instead

一些数据类型要使用Integer.valueOf(“112”)或者Double.valueOf(6.3)之类的，而不要使用new Integer(“112”)这种方式定义

### Cloneable Not Implemented Correctly (CN)

这种模式检查类是否正确实现了Cloneable接口。最常见的错误是调用构造函数分配一个新对象，而不是调用super.clone()。这就导致无法正常地从类继承子类，因为从子类中调用clone 不会返回其类型的实例。

### OBL\_UNSATISFIED\_OBLIGATION: Method may fail to clean up stream or resource

方法可能关闭流或者资源失败，这种情况一般要使用try、catch包起来，使用完后关闭流或者资源，然后在finally再检查一次

例如： try{

…..

rs.close();

} **catch** (Exception e) {

log.info("出错:" + e.getMessage());

} **finally** {

**try** {

**if** (rs != **null**) {

rs.close();

}

} **catch** (HibernateException e) {

log.info("rs.close()出错:" + e.getMessage());

} **catch** (SQLException e) {

log.info("rs.close()出错:" + e.getMessage());

}

}

### Double Checked Locking (DC)

两次检查对象是否为空来保证同步仅对顺序一致性内存模型有效，关于DCL已不再有效的声明发表在http://www.cs.umd.edu/users/pugh/java/memoryModel/DoubleCheckedLocking.html

// Correct multithreaded version

class Foo {

  private Helper helper = null;

 public synchronized Helper getHelper() {

    if (helper == null)

        helper = new Helper();

    return helper;

    }

 // other functions and members...

 }

为了避免在helper已被分配后仍需要同步，提出了DCL，如下

public Helper getHelper() {

 if (helper == null)

 synchronized(this) {

        if (helper == null) helper = new Helper();

 }

return helper;

 }

但这段代码在编译器优化或者共享内存的多处理器上却会出现问题。

最主要的原因就是new Helper方法进行的初始化和向helper中的写入是可以乱序执行的。因此一个线程可能会访问到helper对象，但是helper对象里是默认的初始化值，而非通过构造函数指定。

在java内存模型中，要修正这个故障很容易，只要将helper声明为volatile即可。代码如下：

class Foo {

private volatile Helper helper = null;

public Helper getHelper() {

if (helper == null) {

synchronized(this) {

if (helper == null)

helper = new Helper();

}

} return helper;

}

}

### Dropped Exception (DE)

这个检测器查找try-catch块中，catch语句内容为空，异常被忽略。这种情况的出现是因为程序员认为该种异常不会出现，但是实际执行中一旦出现该异常，就会导致相当严重的后果。

### Suspicious Equals Comparison (EC)

该检测器使用intraprocedural dataflow analysis以确定两个已知不可比较的对象使用了equals()方法进行了比较。这样的比较应一直返回false,并且往往是由于比较了错误的对象，才导致出现该故障。

### Bad Covariant Definition of Equals (Eq)

Java类可能会覆盖equals(Object)方法来定义对象是否等同。编程者有时会使用自己的类作为equals()函数的参数。如：

public boolean equals(Foo obj) {...}

这样的equals变种并没有覆盖Object中的equals方法，就会导致运行时所不期望的行为，尤其是当该类和标准类集合中的类混合使用时，标准类期望equals(Object)被正确覆盖。这种bug不易发现，因为当该类以类的引用方式（而不是超类）访问时，不会发生任何错误。

### Equal Objects Must Have Equal Hashcodes (HE)

Java对象要能够保存在HashMaps和HashSets中，就要同时实现equals(Object)和hashCode()方法。其中很重要的隐含规则就是：经比较相等的对象必须具有相同的hashcode。

考虑下面一种情况，有一个类覆盖了equals方法，但没有覆盖hashCode()方法。而Object类中缺省实现的hashCode()返回一个由java虚拟机随机分配的一个值。这样就可能会导致该类的两个对象equal，但具有不同的hashcode.由于hashcode不同，它们就会被分配到不同的buckets,从而使得在同一hash数据结构中含有两个相同(equal)的对象，违反了HashMap和HashSet的规定。

### Static Field Modifiable By Untrusted Code (MS)

这种bug主要针对的情况为不可信的代码被允许修改static数据，因此就修改了影响所有用户的库的行为。下面是几种不当授权可能发生的情况：

Static但不是final的数据允许public或者protected的访问

Static final的数据允许public或者protected的访问，并且引用到一个可变（更）结构，如数组或者Hashtable

一个方法返回的引用指向static但可变更的数据结构如数据或者Hashtable。

这种故障主要是由于对static所起作用假设不当引起的，static成员的作用是允许独立于类的任何对象使用，无需一个特殊的实例，往往关乎全局的设置。但该数据是否可以修改则未作限制。上述第一种情况是显而易见的，未声明为final的成员允许public访问，就打开了不可信代码向对象实施变更的大门。后两种情况也类似，均是允许对自身进行修改造成漏洞。值得注意的是第二种情况，虽然声明了final，但它的引用却指向一个可变结构，本质上仍允许修改。

### Null Pointer Dereference (NP), Redundant Comparison to Null (RCN)

使用空指针调用函数或者访问变量会导致NullPointerException。探测器使用严格的过程内数据流分析，并且将if语句考虑在内，如if(foo == null){…}, 那么探测器就知道foo在if语句体内为空。

探测器产生两种警告：full statement coverage（？）下一定会出现的空指针为高优先级，full branch coverage下可能出现的空指针为低优先级。实例如下：

// Eclipse 3.0,

// org.eclipse.jdt.internal.ui.compare,

// JavaStructureDiffViewer.java, line 131

Control c= getControl();

if (c == null && c.isDisposed())

return;

另外，该探测器还查找引用的比较，但其中一个或者两个变量值为空。虽然这种比较不会引起运行时异常，但它很可能来自于编程者的错误，也很有可能是由其它错误引起的。如：if (m.parent != this) { //访问了m中的parent,这里的前提即为m非空

add(m);

}

helpMenu = m;

if (m != null) { // 检查m是否为空，与上面的前提相悖

论文中的RCN模式仅扫描一个值在被检测判断是否为空前是否被使用。

### Non-Short-Circuit Boolean Operator (NS)

与C和C++中的操作符一样，&&和||都具有short-circuit evaluation属性(short-circuit evaluation是指，对&&来说如果前件为假那么就不再检查后件而直接进行跳转)。但是&和|则不具有这种性质，当错误的使用了&和|时，就会导致前后件都要无条件地进行判断，从而发生空指针异常。

### Open Stream

当程序打开一个输入或者输出流时，应该保证当它不再可达时被关闭。虽然finalizer函数保证了IO流会在垃圾回收时自动关闭，但可能并不及时。由于操作系统中的文件描述符是有限的资源，如果用光就会导致系统失效。此外，如果一个缓存的输出流没有被关闭，那么相应的数据有可能就不会写到文件里，这是因为java中finalizer并不保证在程序退出时一定会运行。探测器也使用dataflow analysis该故障实例如：

// DrJava stable-20040326

// edu.rice.cs.drjava.ui

// JavadocFrame.java, line 103

private static File \_parsePackagesFile(

File packages, File destDir) {

try {

FileReader fr =

new FileReader(packages);

BufferedReader br =

new BufferedReader(fr);

...

// fr/br are never closed

### Read Return Should Be Checked (RR)

Java.io.InputStream中的read函数负责将多个字节读到缓冲区。由于所指定的字节数有可能大于实际读到的字节数，read方法通过返回一个整型变量来表明本次读取了多少个字节。程序员有时会假设该方法每次都返回所指定的字节数，这样就使得程序读取到stale的数据，并且导致输入流中数据发生混乱。

探测器可以使用数据流分析。一种更直接的方式为，扫描字节码，如果read方法后紧跟着POP指令，那么就给出提示。

### Return Value Should Be Checked (RV)

Java标准库里有很多不可变类，如String就是其中之一。与C++中不同，改变String对象的方法是通过返回一个新String对象的方式来实现修改。因此，程序员有可能会产生误解，忽略了不可变对象上方法调用的返回值。

检测器的实现与Read Return模式类似，在字节码级，查找一组特定方法的调用，其后紧跟着POP或者POP2指令。这里的特定方法包括：

• Any String method returning a String

• StringBuffer.toString()

• Any method of InetAddress, BigInteger, or BigDecimal

• MessageDigest.digest(byte[])

• The constructor for any subclass of Thread or Throwable.

### Non-serializable Serializable class (SE)

该模式寻找无法被序列化但却实现了Serializable接口的类。这么做的原因为：

它包含了一个非临时性的变量，该变量的类型并没有实现Serializable。该类的超类无法序列化，并且没有可访问的无参构造函数。

### Uninitialized Read In Constructor (UR)

构造对象时，所有的变量都被置为其类型的默认值。通常来说，在变量写入之前读取其值是没有意义的，因此，需要检测对象的构造函数来查找是否含有未被赋值即被读取的变量。该bug经常是由于程序员弄错了变量名（类似的）引起的。实例如下：

// JBoss 4.0.0RC1

// org.jboss.monitor,

// SnapshotRecordingMonitor.java,

// line 44

public SnapshotRecordingMonitor()

{

log = Logger.getLogger(monitorName);

history = new ArrayList(100);

}

### Unconditional Wait (UW)

在多线程环境中，错误地使用wait()和notify()来调度线程是常见错误。该检测模式寻找在进入同步块时无条件地wait()。一般的，这表明与wait相关的条件判断是在无锁（不在同步块内）情况下进行的，有可能导致其它线程的notification被忽略。

该模式检测器使用字节码扫描，查找wait()调用前紧跟着为monitorenter的指令，并且没有分支指令跳转到该wait调用。实例如下：

// JBoss 4.0.0RC1

// org.jboss.deployment.scanner

// AbstractDeploymentScanner.java, line 185

// If we are not enabled, then wait

if (!enabled) { // 条件判断不在同步块内

try {

synchronized (lock) {

//if (!enabled) 应该在这里

lock.wait();

...

### Wait Not In Loop (Wa)

实现条件wait的最健壮方式为在一个同步块内进行循环判断，当条件为假时调用wait。程序员经常认为这个循环是不必要的，比如可能只有一个与此关联的条件，因此所有的notification都对应着该条件为真。但即使仅关联一个条件，在wait被唤醒到获得锁之间还有一段时间，在这段时间里其它线程有可能将该条件置为假。

该模式的检测器记录第一次wait()调用和第一次分支，如果wait在最早的分支之前，那么就发出警告。